Report_Reproducibility_and_Extensions_Orujlu

August 13, 2023

1 Reproducibility Report

1.0.1 Student : Omar Faig Orujlu(03750822)

1.0.2 Paper : Associative Memory in Iterated Overparameterized Sigmoid Autoencoders

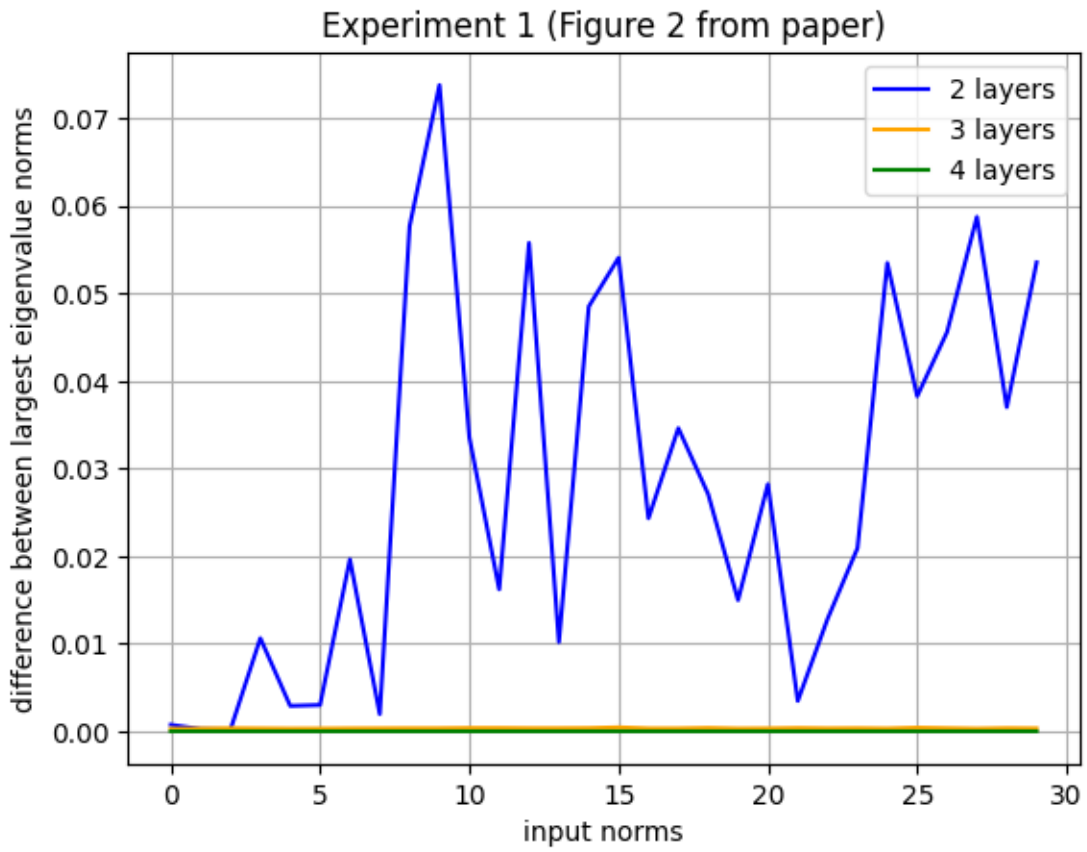
2 Experiment 1 - Single Training Example

In this experiment the same results as in Figure 2 in the paper is reproduced. As stated in the paper the largest (Operator) norm of initial Jacobian for 2 layer sigmoid network is concentrated around 1/2. Also , the initial and final Jacobians are very similar (almost equal) as the number of layers are increased. In the below diagram the results for 3 and 4 layers are the same as in the paper and for 2 layer case the theoretical results are reproduced .

[3]:
```
import copy
import numpy as np
import torch.nn.functional as F
import torch
import torch.nn as nn
import torchvision
import matplotlib.pyplot as plt
from torch import linalg as LA
from torchsummary import summary
from torch.nn.utils import weight_norm
import utils
from matplotlib.ticker import PercentFormatter

#*********************************************************************************
diff_list_2=[]
diff_list_3=[]
diff_list_4=[]
epochs = 0
outputs = []
losses = []
for k in range(2,5):
        if k == 2:
            for i in range(1,31):
                input = utils.sample_input_(i,dim=32)
```

```
                model = utils.
↪Autoencoder_2_layers_(input_dim=32,hidden_dim=1000)
                optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                initial_state_dict = copy.deepcopy(model.state_dict())
                loss_function = nn.MSELoss()
                while True:
                    epochs+=1
                    reconstructed = model(input)
                    loss = loss_function(reconstructed, input)
                    optimizer.zero_grad()
                    loss.backward()
                    losses.append(loss.detach().numpy())
                    outputs.append((epochs, input, reconstructed))
                    optimizer.step()
                    if loss<1e-7 :
                        trained_state_dict = model.state_dict()
                        _,diff_2 = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,input)
                        diff_list_2.append(diff_2)
                        break

            k=+1
        if k == 3:
                for i in range(1,31):
                    input = utils.sample_input_(i,dim=32)
                    model = utils.
↪Autoencoder_3_layers(input_dim=32,hidden_dim=1000)
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:
                            trained_state_dict = model.state_dict()
                            diff_list_3.append(utils.
↪calculate_jacobian_3_layers(initial_state_dict,trained_state_dict,input))
                            break
                k=+1
        if k == 4:
                for i in range(1,31):
```

```
                    input = utils.sample_input_(i,dim=32)
                    model = utils.
 ↪Autoencoder_4_layers(input_dim=32,hidden_dim=1000)
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:
                            trained_state_dict = model.state_dict()
                            diff_list_4.append(utils.
 ↪calculate_jacobian_4_layers(initial_state_dict,trained_state_dict,input))
                            break
                k=+1

plt.plot(diff_list_2,color="blue",label="2 layers")
plt.plot(diff_list_3,color="orange",label="3 layers")

plt.plot(diff_list_4,color="green",label="4 layers")
plt.xlabel('input norms')
plt.ylabel('difference between largest eigenvalue norms')
plt.title('Experiment 1 (Figure 2 from paper)')
#plt.xlim(-1,32)

plt.grid()
plt.legend()
plt.show()
```



3 Experiment 2 - Multiple Training Example

3.0.1 Linear Region

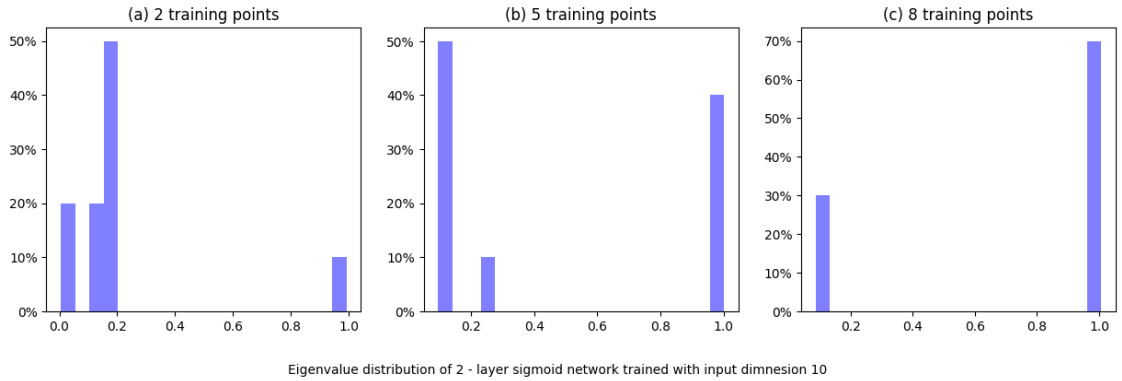
This experiment is based on Section 5.3 where the eigenvalue distribution in linear region is demonstrated. Here, a 2 layer sigmoid autoencoder is trained with 2,5, and 8 points and these points have input radius of 1. According to lemma 4 of the paper there should be n-1 eigenvalues with norm around 1. This can be seen in the generated(reproduced) diagram , where 10%,40% and 70% of all eigenvalues are near 1. Since there is eigenvalues with norm 1, it can also be concluded that network operates in linear region

[8]:
```
x_2_points = utils.generate_x_training_points(1,10,2)
x_5_points = utils.generate_x_training_points(1,10,5)
x_8_points = utils.generate_x_training_points(1,10,8)
for i in range(0,3):
    if i ==0:
        model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=1000)
        optimizer = torch.optim.SGD(params=model.parameters(),lr=3)
        initial_state_dict = copy.deepcopy(model.state_dict())
        loss_function = nn.MSELoss()
```

```
        epochs = 0
        outputs = []
        losses = []
        while True:
            epochs+=1
            reconstructed = model(x_2_points)
            loss = loss_function(reconstructed, x_2_points)
          #  if epochs%1000==0:
             # print(epochs , ": Current loss:", loss)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.detach().numpy())
            optimizer.step()
            if loss<1e-7:
                trained_state_dict = model.state_dict()
                eigen_values_p1,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_2_points[0])
                eigen_values_p2,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_2_points[1])
                eigen_values    = torch.cat((eigen_values_p2,eigen_values_p1))
                eigen_values_normed_2p = torch.abs(eigen_values)
                break
    i+=1
    if i ==1:
        model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=1000)
        optimizer = torch.optim.SGD(params=model.parameters(),lr=3)
        initial_state_dict = copy.deepcopy(model.state_dict())
        loss_function = nn.MSELoss()
        epochs = 0
        outputs = []
        losses = []
        while True:
            epochs+=1
            reconstructed = model(x_5_points)
            loss = loss_function(reconstructed, x_5_points)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.detach().numpy())
            optimizer.step()
            if loss<1e-7:

                trained_state_dict = model.state_dict()
                eigen_values_p5_1,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[0])
                eigen_values_p5_2,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[1])
```

```
                eigen_values_p5_3,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
                eigen_values_p5_4,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[3])
                eigen_values_p5_5,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[4])
                eigen_values_p5    = torch.
↪cat((eigen_values_p5_1,eigen_values_p5_2,eigen_values_p5_3,eigen_values_p5_4,eigen_values_p
                eigen_values_normed_5p = torch.abs(eigen_values_p5)
                break
    i+=1
    if i ==2:
        model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=1000)
        optimizer = torch.optim.SGD(params=model.parameters(),lr=3)
        initial_state_dict = copy.deepcopy(model.state_dict())
        loss_function = nn.MSELoss()
        epochs = 0
        outputs = []
        losses = []
        while True:
            epochs+=1
            reconstructed = model(x_8_points)
            loss = loss_function(reconstructed, x_8_points)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.detach().numpy())
            outputs.append((epochs, x_8_points, reconstructed))
            optimizer.step()
            if loss<1e-7:
                trained_state_dict = model.state_dict()
                eigen_values_p8_1,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[0])
                eigen_values_p8_2,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[1])
                eigen_values_p8_3,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[2])
                eigen_values_p8_4,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[3])
                eigen_values_p8_5,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[4])
                eigen_values_p8_6,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[5])
                eigen_values_p8_7,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[6])
                eigen_values_p8_8,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_8_points[7])
```

```
                eigen_values_p8    = torch.
 ↪cat((eigen_values_p8_1,eigen_values_p8_2,eigen_values_p8_3,eigen_values_p8_4,eigen_values_p
                eigen_values_normed_8p = torch.abs(eigen_values_p8)
                break

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 4))
axes[0].hist(eigen_values_normed_2p,weights=np.
 ↪ones(len(eigen_values_normed_2p)) / len(eigen_values_normed_2p), bins=20,␣
 ↪color='blue', alpha=0.5)
axes[1].hist(eigen_values_normed_5p,weights=np.
 ↪ones(len(eigen_values_normed_5p)) / len(eigen_values_normed_5p), bins=20,␣
 ↪color='blue', alpha=0.5)
axes[2].hist(eigen_values_normed_8p,weights=np.
 ↪ones(len(eigen_values_normed_8p)) / len(eigen_values_normed_8p), bins=20,␣
 ↪color='blue', alpha=0.5)

#plt.hist(eigens, weights=np.ones(len(eigens)) / len(eigens),bins=70)
#plt.hist(eig, weights=np.ones(len(eig)) / len(eig),bins=70)
axes[0].yaxis.set_major_formatter(PercentFormatter(1))
axes[1].yaxis.set_major_formatter(PercentFormatter(1))
axes[2].yaxis.set_major_formatter(PercentFormatter(1))
# Set titles for each subplot
axes[0].set_title('(a) 2 training points')
axes[1].set_title('(b) 5 training points')
axes[2].set_title('(c) 8 training points')

# Set common y-axis label
fig.text(0.3, -0.05, 'Eigenvalue distribution of 2 - layer sigmoid network␣
 ↪trained with input dimnesion 10', va='center', rotation='horizontal')
plt.show()
```



3.1 Beyond Linear Region

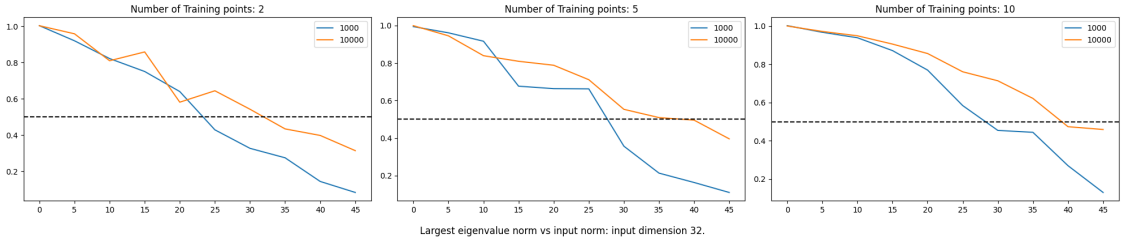
Here the results from Figure 3 is reproduced. However, in the paper authors have experimented with 5,20, and 40 training points with hidden layer size from 1000 to 1000000. In the figure below, the results for 2,5, and 10 points for hidden layers size 1000 and 10000 are depicted.That's why results could not be reproduced exactly, however, similar behavior can be observed. Note: I tried to run the experiment with 20 training points but it took 30+ hours(for input norm 1.0).

[5]:
```
hidden_layers=[1000,10000]
results_1={}
results_2={}
results_3={}
for t in range(0,3):
    if t==0:
        epochs = 0
        outputs = []
        losses = []

        for k in range(0,len(hidden_layers)):
            largest_eigen_list=[]
            for i in range (3,53):
                    x_2_points = utils.
↪generate_x_training_points(i,32,2)#increase i too hundred and generate␣
↪points at i/2 norms
                    model = utils.
↪Autoencoder_2_layers_(input_dim=32,hidden_dim=hidden_layers[k])
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=5)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    epochs = 0
                    outputs = []
                    losses = []
                    while True:
                        epochs+=1
                        reconstructed = model(x_2_points)
                        loss = loss_function(reconstructed, x_2_points)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        optimizer.step()
                        if loss<1e-7:#should be 1e-7 according to paper
                            trained_state_dict = model.state_dict()
                            eigen_values_p3,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_2_points[2])
                            larg_eigen=torch.max(torch.abs(eigen_values_p3))
                            largest_eigen_list.append(larg_eigen)
                            break
```

```
            results_1[k]=largest_eigen_list
    if t==1:
        epochs = 0
        outputs = []
        losses = []
        for k in range(0,len(hidden_layers)):
            largest_eigen_list=[]
            for i in range (3,53):
                    x_5_points = utils.
↪generate_x_training_points(i,32,5)#increase i too hundred and generate␣
↪points at i/2 norms
                    model = utils.
↪Autoencoder_2_layers_(input_dim=32,hidden_dim=hidden_layers[k])
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=5)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    epochs = 0
                    outputs = []
                    losses = []
                    while True:
                        epochs+=1
                        reconstructed = model(x_5_points)
                        loss = loss_function(reconstructed, x_5_points)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        optimizer.step()
                        if loss<1e-7:#should be 1e-7 according to paper
                            trained_state_dict = model.state_dict()
                            eigen_values_p3,_ = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
                            larg_eigen=torch.max(torch.abs(eigen_values_p3))
                            largest_eigen_list.append(larg_eigen)
                            break
            results_2[k]=largest_eigen_list
    if t==2:
        epochs = 0
        outputs = []
        losses = []
        for k in range(0,len(hidden_layers)):
            largest_eigen_list=[]
            for i in range (3,53):
                    x_10_points = utils.
↪generate_x_training_points(i,32,10)#increase i too hundred and generate␣
↪points at i/2 norms
                    model = utils.
↪Autoencoder_2_layers_(input_dim=32,hidden_dim=hidden_layers[k])
```

```
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=5)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    epochs = 0
                    outputs = []
                    losses = []
                    while True:
                        epochs+=1
                        reconstructed = model(x_10_points)
                        loss = loss_function(reconstructed, x_10_points)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        optimizer.step()
                        if loss<1e-7:#should be 1e-7 according to paper
                            trained_state_dict = model.state_dict()
                            eigen_values_p3,_ = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_10_points[2])
                            larg_eigen=torch.max(torch.abs(eigen_values_p3))
                            largest_eigen_list.append(larg_eigen)
                            break
            results_3[k]=largest_eigen_list

x_values = list(range(0, len(results_1[1]), 5))
fig, axs = plt.subplots(1, 3, figsize=(20, 4))

# Plotting the line graphs for each dictionary
axs[0].plot(x_values, results_1[0][::5], label='1000')
axs[0].plot(x_values, results_1[1][::5], label='10000')
axs[0].set_title('Number of Training points: 2')
axs[0].set_xticks(x_values)
axs[0].legend()

axs[1].plot(x_values, results_2[0][::5], label='1000')
axs[1].plot(x_values, results_2[1][::5], label='10000')
axs[1].set_title('Number of Training points: 5')
axs[1].set_xticks(x_values)
axs[1].legend()

axs[2].plot(x_values, results_3[0][::5], label='1000')
axs[2].plot(x_values, results_3[1][::5], label='10000')
axs[2].set_title('Number of Training points: 10')
axs[2].set_xticks(x_values)
axs[2].legend()

# Adding the horizontal line to each diagram
axs[0].axhline(0.5, linestyle='--', color='black')
```

```
axs[1].axhline(0.5, linestyle='--', color='black')
axs[2].axhline(0.5, linestyle='--', color='black')

fig.text(0.5, -0.05, 'Largest eigenvalue norm vs input norm: input dimension 32.
  ↪', ha='center', fontsize=12)

plt.tight_layout()
plt.show()
```



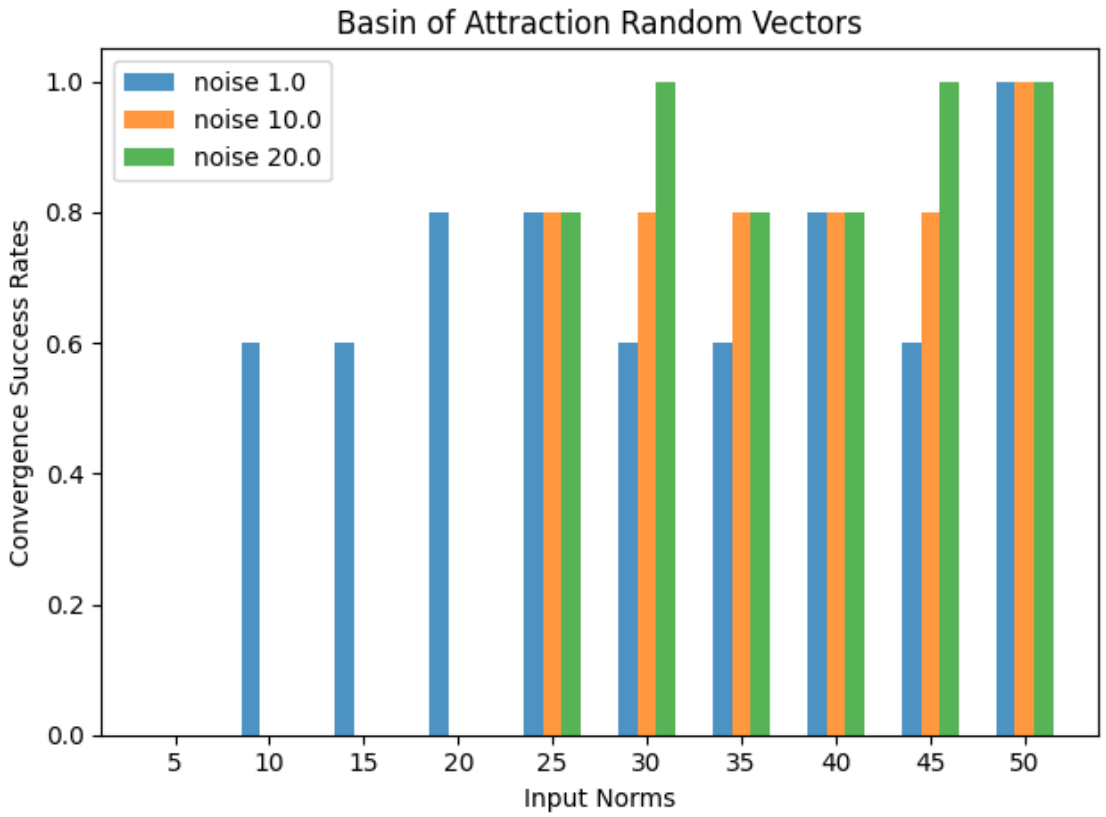
4 Experiment 4 - Basin of Attraction

Here the results from section 5.4 are reproduced. 5 training points with dimension 32 are used for training the model.. As stated in the paper, attractor formulation in small norms fails, this can also be observed in the following diagrams.

[6]:
```
epochs = 0
outputs = []
losses = []
input_norms = list(range(5, 51, 5))
std_list = [1, 10, 20]
conver = {}
for t in std_list:
    for i in input_norms:
        input = utils.generate_x_training_points(i, 32, 5)
        model = utils.Autoencoder_2_layers_(input_dim=32, hidden_dim=10000)
        optimizer = torch.optim.SGD(params=model.parameters(), lr=3)
        loss_function = nn.MSELoss()
        while True:
            epochs += 1
            reconstructed = model(input)
            loss = loss_function(reconstructed, input)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.detach().numpy())
            #outputs.append((epochs, input, reconstructed))
            optimizer.step()
            if loss < 1e-7:
```

```
                noisy = input + np.sqrt(t) * torch.randn_like(input)
                #print("original input",input)

                #print("original noisy output",noisy)
                for k in range(0, 51):
                    noisy = model(noisy)
                    #noisy = noisy_result
                   # print("noisy output after iteration no ",k, "noisys",noisy)
                    loss_ = loss_function(noisy, input)
                    #print("loss_ of convergence", loss_, "num of iteration",␣
 ↪k, "norm", i, "noise", t)
                    if k==50 and  not (loss_<1e-2):
                        convergence_rate = 0
                        conver.setdefault(t, []).append(convergence_rate)
                        #print("converged", 0 , "after iteration no ", k, "norm␣
 ↪of vector", i, "convergence rate", convergence_rate, "noise", t)
                        break
                    if loss_ < 1e-2 :
                        convergence = 0
                        convergence_rate=0
                        #c#onverged_samples = torch.sum(torch.all(torch.
 ↪abs(input - noisy) < 0.1, dim=1)).item()
                        #convergence_rate = converged_samples / input.shape[0]
                        if loss_function(noisy[0], input[0]) < 0.01:␣
 ↪convergence += 1
                        if loss_function(noisy[1], input[1]) < 0.01:␣
 ↪convergence += 1
                        if loss_function(noisy[2], input[2]) < 0.01:␣
 ↪convergence += 1
                        if loss_function(noisy[3], input[3]) < 0.01:␣
 ↪convergence += 1
                        if loss_function(noisy[4], input[4]) < 0.01:␣
 ↪convergence += 1
#
                        convergence_rate = convergence / len(input)
                        conver.setdefault(t, []).append(convergence_rate)
                        #print("converged", convergence, "after iteration no ",␣
 ↪k, "norm of vector", i, "convergence rate", convergence_rate, "noise", t)
                        break
                break
x_labels = list(range(5, 51, 5))
data1 = conver[1]
data2 = conver[10]
data3 = conver[20]

# Plotting
```

```
x = range(len(x_labels))
bar_width = 0.2
opacity = 0.8

plt.bar(x, data1, bar_width, alpha=opacity, label='noise 1.0')
plt.bar([val + bar_width for val in x], data2, bar_width, alpha=opacity,␣
 ↪label='noise 10.0')
plt.bar([val + 2 * bar_width for val in x], data3, bar_width, alpha=opacity,␣
 ↪label='noise 20.0')

plt.xlabel('Input Norms')
plt.ylabel('Convergence Success Rates')
plt.title('Basin of Attraction Random Vectors')
plt.xticks([val + bar_width for val in x], input_norms)
plt.legend()
plt.tight_layout()
plt.show()
```



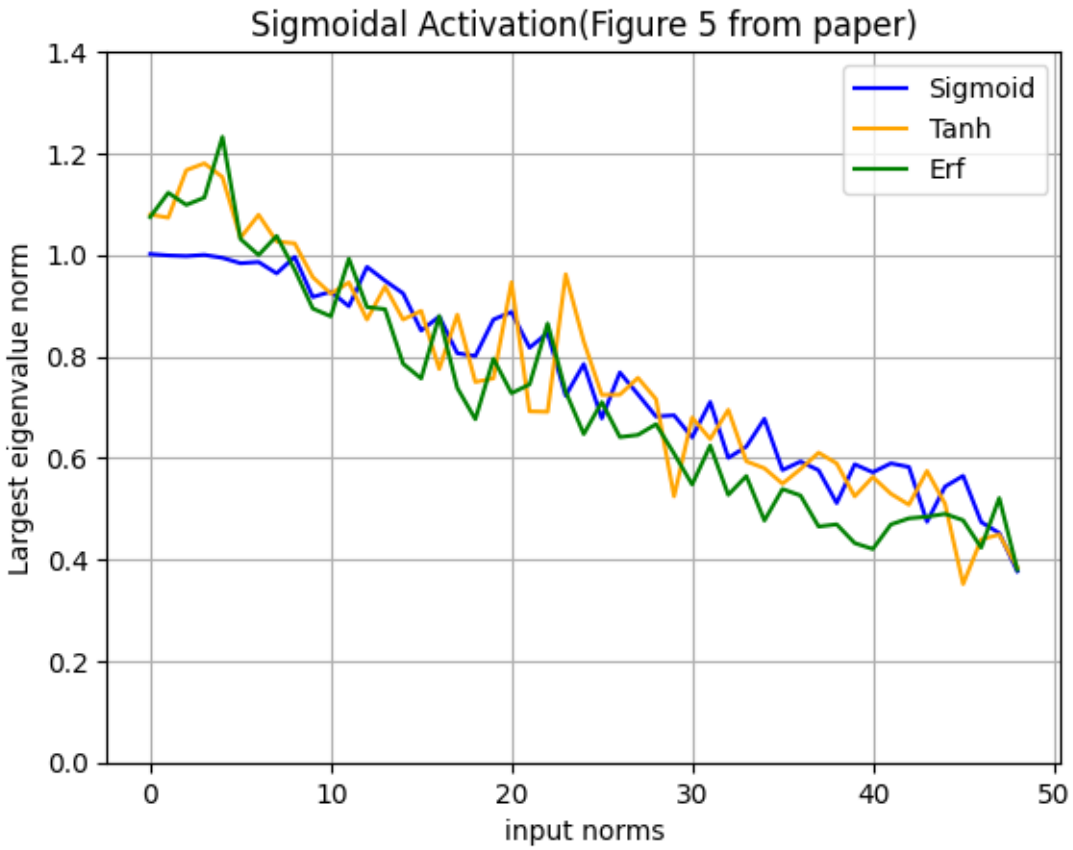
5 Experiment 5 - Sigmoidal Activation

Here the results from Section 5.5 are reproduced. A two layer sigmoid, tanh and erf autoencoder are trained. However, here sigmoid network has 10 training points and others 20. Results are very similar to the ones in figure 5. in the paper and as it is stated in paper , erf and tanh have eigenalues bigger than 1.0 at small input norms.

[7]:
```
largest_eigen_list_sigmoid=[]
largest_eigen_list_tanh=[]
largest_eigen_list_erf=[]
epochs = 0
outputs = []
losses = []
for i in range (2,51):
            x_5_points = utils.generate_x_training_points(i,32,20)
            model = utils.
 ↪Autoencoder_2_layers_erf(input_dim=32,hidden_dim=10000)
            optimizer = torch.optim.SGD(params=model.parameters(),lr=5)
            initial_state_dict = copy.deepcopy(model.state_dict())
            loss_function = nn.MSELoss()
            epochs = 0
            outputs = []
            losses = []
            while True:
                epochs+=1
                reconstructed = model(x_5_points)
                loss = loss_function(reconstructed, x_5_points)

                optimizer.zero_grad()
                loss.backward()
                losses.append(loss.detach().numpy())
                optimizer.step()
                if loss<1e-7:
                    trained_state_dict = model.state_dict()
                    eigen_values_p3,_ = utils.
 ↪calculate_jacobian_2_layers_erf(initial_state_dict,trained_state_dict,x_5_points[2])
                    larg_eigen=torch.max(torch.abs(eigen_values_p3))
                    largest_eigen_list_erf.append(larg_eigen)
                    break

for i in range (2,51):
            x_5_points = utils.generate_x_training_points(i,32,10)
            model = utils.Autoencoder_2_layers_(input_dim=32,hidden_dim=10000)
            optimizer = torch.optim.SGD(params=model.parameters(),lr=5)
            initial_state_dict = copy.deepcopy(model.state_dict())
            loss_function = nn.MSELoss()
            epochs = 0
```

```
            outputs = []
            losses = []
            while True:
                epochs+=1
                reconstructed = model(x_5_points)
                loss = loss_function(reconstructed, x_5_points)
                optimizer.zero_grad()
                loss.backward()
                losses.append(loss.detach().numpy())
                optimizer.step()
                if loss<1e-7:
                    trained_state_dict = model.state_dict()
                    eigen_values_p3,_ = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
                    larg_eigen=torch.max(torch.abs(eigen_values_p3))
                    largest_eigen_list_sigmoid.append(larg_eigen)
                    break

for i in range (2,51):
            x_5_points = utils.generate_x_training_points(i,32,20)
            model = utils.
 ↪Autoencoder_2_layers_tanh(input_dim=32,hidden_dim=10000)
            optimizer = torch.optim.SGD(params=model.parameters(),lr=5)
            initial_state_dict = copy.deepcopy(model.state_dict())
            loss_function = nn.MSELoss()
            epochs = 0
            outputs = []
            losses = []
            while True:
                epochs+=1
                reconstructed = model(x_5_points)
                loss = loss_function(reconstructed, x_5_points)
                optimizer.zero_grad()
                loss.backward()
                losses.append(loss.detach().numpy())
                optimizer.step()
                if loss<1e-7:
                    trained_state_dict = model.state_dict()
                    eigen_values_p3,_ = utils.
 ↪calculate_jacobian_2_layers_tanh(initial_state_dict,trained_state_dict,x_5_points[2])
                    larg_eigen=torch.max(torch.abs(eigen_values_p3))
                    largest_eigen_list_tanh.append(larg_eigen)
                    break

plt.plot(largest_eigen_list_sigmoid,color="blue",label="Sigmoid")
plt.plot(largest_eigen_list_tanh,color="orange",label="Tanh")
plt.plot(largest_eigen_list_erf,color="green",label="Erf")
```

```
plt.xlabel('input norms')
plt.ylabel('Largest eigenvalue norm')
plt.title('Input Radius and Eigenvalue Norm Curve for Different Activation␣
 ↪Functions')
plt.ylim(0,1.4)
plt.grid()
plt.legend()
plt.show()
```



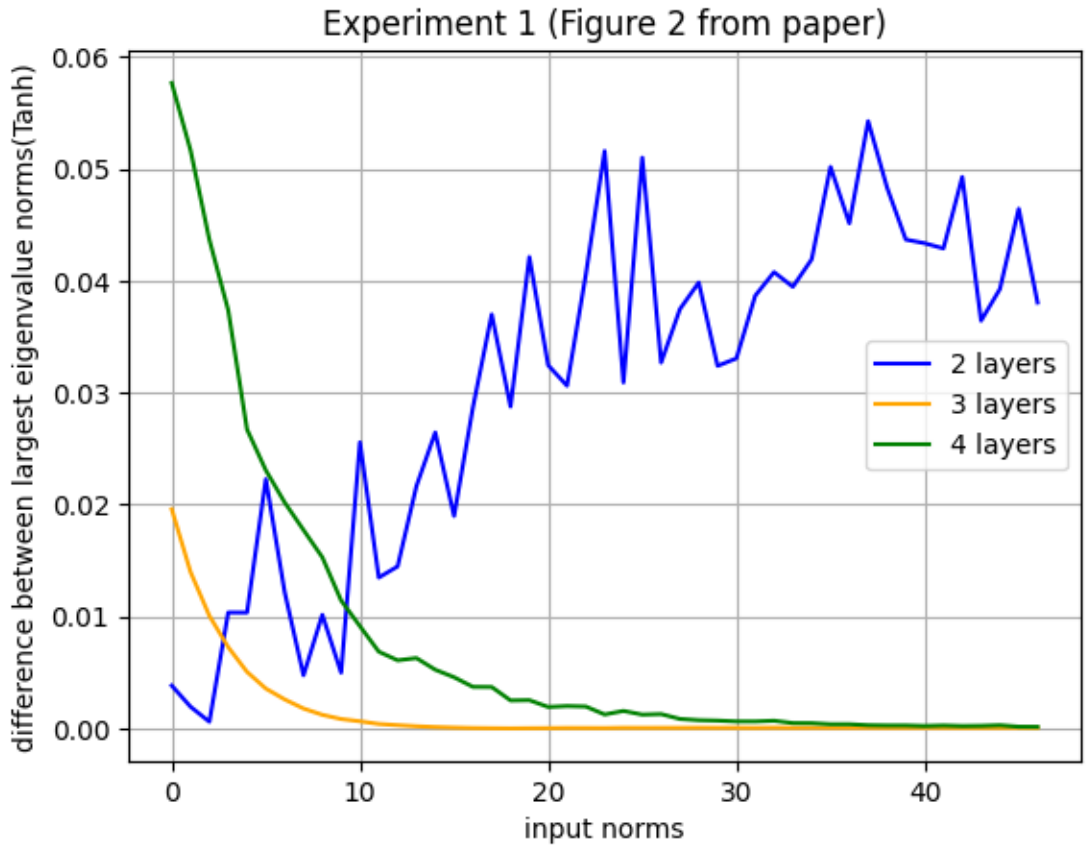
6 Extension 1 - Lipschnitz and Non-Lipschnitz activation functions

In the paper , authors argue that same/similar results can be obtained if tanh/erf functions are used for above experiments.That's why I repeated the first experiment from the paper with Lipschitz(TanH) and non-Lipschitz (ReLU) activation functions. The results are demonstrated below. Here, it can be observed that the Tanh gives similar results, however, the results with ReLU is quite ambigous. Because, in 2 and 3 layer cases the differences are small and in 4 layers case the difference is larger. So , for having attractor behavior in a network Lipschitz activation function is important.

[2]:
```
diff_list_2=[]
diff_list_3=[]
diff_list_4=[]
epochs = 0
outputs = []
losses = []
for k in range(2,5):
        if k == 2:
            for i in range(3,50):
                epochs = 0
                input = utils.sample_input_(i,dim=32)
                model = utils.
↪Autoencoder_2_layers_tanh(input_dim=32,hidden_dim=1000)
                optimizer = torch.optim.SGD(params=model.parameters(),lr=3)
                initial_state_dict = copy.deepcopy(model.state_dict())
                loss_function = nn.MSELoss()
                while True:
                    epochs+=1
                    reconstructed = model(input)
                    loss = loss_function(reconstructed, input)
                    optimizer.zero_grad()
                    loss.backward()
                    losses.append(loss.detach().numpy())
                    #outputs.append((epochs, input, reconstructed))
                    optimizer.step()
                    if loss<1e-7 :
                        trained_state_dict = model.state_dict()
                        _,diff_2 = utils.
↪calculate_jacobian_2_layers_tanh(initial_state_dict,trained_state_dict,input)
                        diff_list_2.append(diff_2)
                        break

            k=+1
        if k == 3:
                for i in range(3,50):
                    epochs = 0
                    input = utils.sample_input_(i,dim=32)
                    model = utils.
↪Autoencoder_3_layers_tanh(input_dim=32,hidden_dim=1000)
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
```

```
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        optimizer.step()

                        if loss<1e-7:

                            trained_state_dict = model.state_dict()
                            diff_list_3.append(utils.
 ↪calculate_jacobian_3_layers_tanh(initial_state_dict,trained_state_dict,input))
                            break
                k=+1
        if k == 4:
                for i in range(3,50):
                    epochs = 0
                    input = utils.sample_input_(i,dim=32)
                    model = utils.
 ↪Autoencoder_4_layers_tanh(input_dim=32,hidden_dim=1000)
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        #outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:

                            trained_state_dict = model.state_dict()
                            diff_list_4.append(utils.
 ↪calculate_jacobian_4_layers_tanh(initial_state_dict,trained_state_dict,input))
                            break
                k=+1

plt.plot(diff_list_2,color="blue",label="2 layers")
plt.plot(diff_list_3,color="orange",label="3 layers")
plt.plot(diff_list_4,color="green",label="4 layers")
plt.xlabel('input norms')
plt.ylabel('difference between largest eigenvalue norms(Tanh)')
plt.title('Experiment 1 (Figure 2 from paper)')
#plt.xlim(-1,32)

plt.grid()
plt.legend()
```

```
plt.show()
```

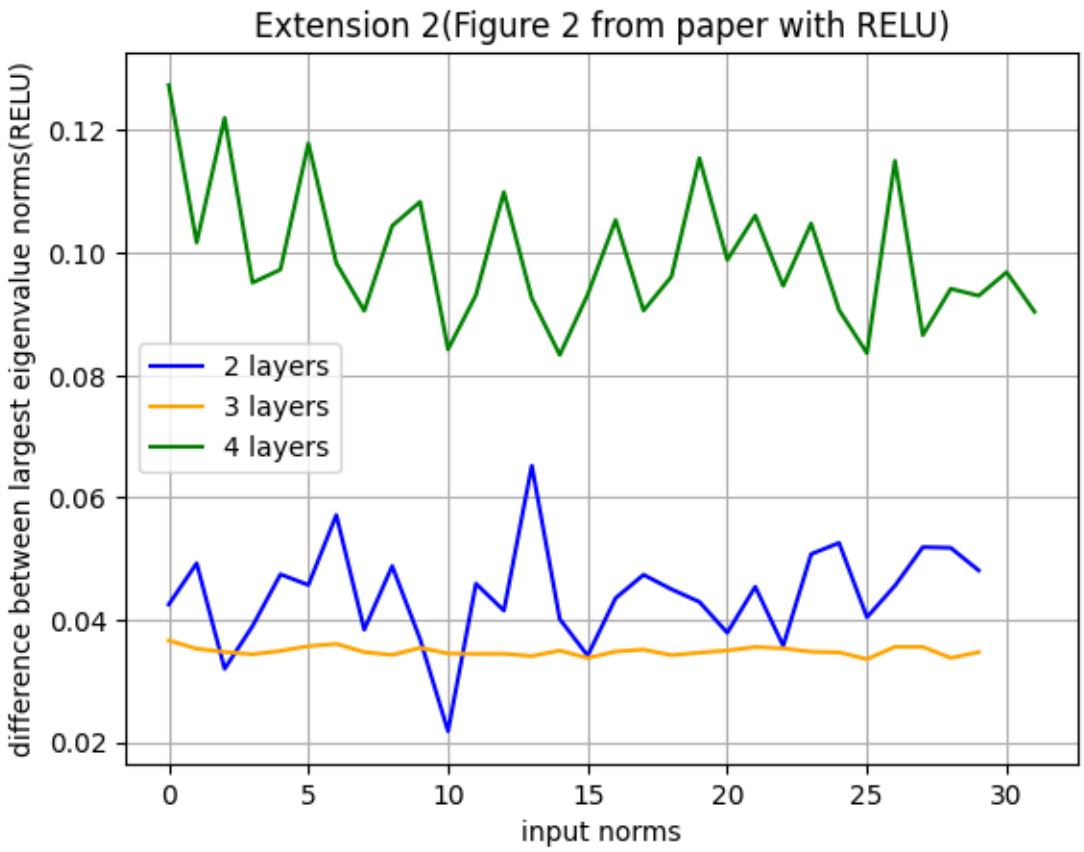


7 RELU

[3]:
```
diff_list_2=[]
diff_list_3=[]
diff_list_4=[]
epochs = 0
outputs = []
losses = []
for k in range(2,5):
        if k == 2:
            for i in range(2,32):
                epochs = 0
                input = utils.sample_input_(i,dim=32)
                model = utils.
 ↪Autoencoder_2_layers_relu(input_dim=32,hidden_dim=1000)
                optimizer = torch.optim.Adam(params=model.parameters(),lr=0.01)
                initial_state_dict = copy.deepcopy(model.state_dict())
```

```
                loss_function = nn.MSELoss()
                while True:
                    epochs+=1
                    reconstructed = model(input)
                    loss = loss_function(reconstructed, input)
                    optimizer.zero_grad()
                    loss.backward()
                    losses.append(loss.detach().numpy())
                    optimizer.step()
                    if loss<1e-7 :
                        trained_state_dict = model.state_dict()
                        _,diff_2 = utils.
↪calculate_jacobian_2_layers_relu(initial_state_dict,trained_state_dict,input)
                        diff_list_2.append(diff_2)
                        break

            k=+1
        if k == 3:
                for i in range(2,32):
                    epochs = 0
                    input = utils.sample_input_(i,dim=32)
                    model = utils.
↪Autoencoder_3_layers_relu(input_dim=32,hidden_dim=1000)
                    optimizer = torch.optim.Adam(params=model.parameters(),lr=0.
↪01)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        #outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:

                            trained_state_dict = model.state_dict()
                            diff_list_3.append(utils.
↪calculate_jacobian_3_layers_relu(initial_state_dict,trained_state_dict,input))
                            break
                k=+1
        if k == 4:
                for i in range(1,33):
                    epochs = 0
                    input = utils.sample_input_(i,dim=32)
```

```
                    model = utils.
↪Autoencoder_4_layers_relu(input_dim=32,hidden_dim=1000)
                    optimizer = torch.optim.Adam(params=model.parameters(),lr=0.
↪01)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        #outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:

                            trained_state_dict = model.state_dict()
                            diff_list_4.append(utils.
↪calculate_jacobian_4_layers_relu(initial_state_dict,trained_state_dict,input))
                            break
                k=+1

plt.plot(diff_list_2,color="blue",label="2 layers")
plt.plot(diff_list_3,color="orange",label="3 layers")
plt.plot(diff_list_4,color="green",label="4 layers")
plt.xlabel('input norms')
plt.ylabel('difference between largest eigenvalue norms(RELU)')
plt.title('Extension 2(Figure 2 from paper with RELU)')
#plt.xlim(-1,32)

plt.grid()
plt.legend()
plt.show()
```



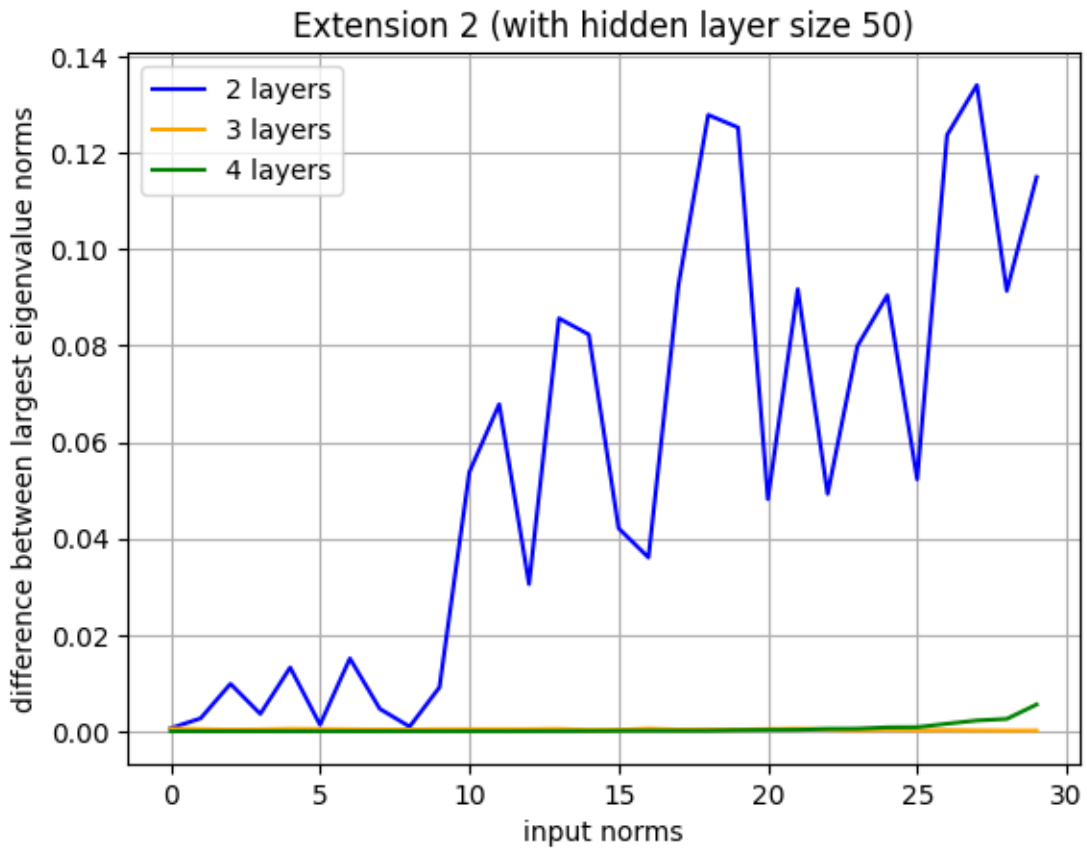
8 Extension 2 - Importance of NTK limit

In order to demonstrate the importance of size of hidden layers the following experiment is conducted. Here the hidden layer is chosen to be 50 instead of 1000 and 1st experiment of the paper is repeated. Here it can be observed that the difference between largest eigenvalues of Jacobians in 2 layer case is almost 2 times higher than the original experiment. Hence , the attractor formation can fail.

[2]:
```
diff_list_2=[]
diff_list_3=[]
diff_list_4=[]
epochs = 0
outputs = []
losses = []
for k in range(2,5):
        if k == 2:
            for i in range(1,31):
                input = utils.sample_input_(i,dim=32)
                model = utils.Autoencoder_2_layers_(input_dim=32,hidden_dim=50)
```

```
                optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                initial_state_dict = copy.deepcopy(model.state_dict())
                loss_function = nn.MSELoss()
                while True:
                    epochs+=1
                    reconstructed = model(input)
                    loss = loss_function(reconstructed, input)
                    optimizer.zero_grad()
                    loss.backward()
                    losses.append(loss.detach().numpy())
                    outputs.append((epochs, input, reconstructed))
                    optimizer.step()
                    if loss<1e-7 :
                        trained_state_dict = model.state_dict()
                        _,diff_2 = utils.
↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,input)
                        diff_list_2.append(diff_2)
                        break

            k=+1
        if k == 3:
                for i in range(1,31):
                    input = utils.sample_input_(i,dim=32)
                    model = utils.
↪Autoencoder_3_layers(input_dim=32,hidden_dim=50)
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:
                            trained_state_dict = model.state_dict()
                            diff_list_3.append(utils.
↪calculate_jacobian_3_layers(initial_state_dict,trained_state_dict,input))
                            break
                k=+1
        if k == 4:
                for i in range(1,31):
                    input = utils.sample_input_(i,dim=32)
```

```
                    model = utils.
 ↪Autoencoder_4_layers(input_dim=32,hidden_dim=50)
                    optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
                    initial_state_dict = copy.deepcopy(model.state_dict())
                    loss_function = nn.MSELoss()
                    while True:
                        epochs+=1
                        reconstructed = model(input)
                        loss = loss_function(reconstructed, input)
                        optimizer.zero_grad()
                        loss.backward()
                        losses.append(loss.detach().numpy())
                        outputs.append((epochs, input, reconstructed))
                        optimizer.step()
                        if loss<1e-7:
                            trained_state_dict = model.state_dict()
                            diff_list_4.append(utils.
 ↪calculate_jacobian_4_layers(initial_state_dict,trained_state_dict,input))
                            break
                k=+1

plt.plot(diff_list_2,color="blue",label="2 layers")
plt.plot(diff_list_3,color="orange",label="3 layers")
plt.plot(diff_list_4,color="green",label="4 layers")
plt.xlabel('input norms')
plt.ylabel('difference between largest eigenvalue norms')
plt.title('Extension 2 (with hidden layer size 50)')
#plt.xlim(-1,32)

plt.grid()
plt.legend()
plt.show()
```



9 Extension 3 - Attractor vs Generalization

In the conclusion part of the paper the authors point out to the further research on the relation between attractor formation and generalization. Here following experiments are conducted.

9.1 Hidden Size 1000 - Input norm 15 - 100 Training Points

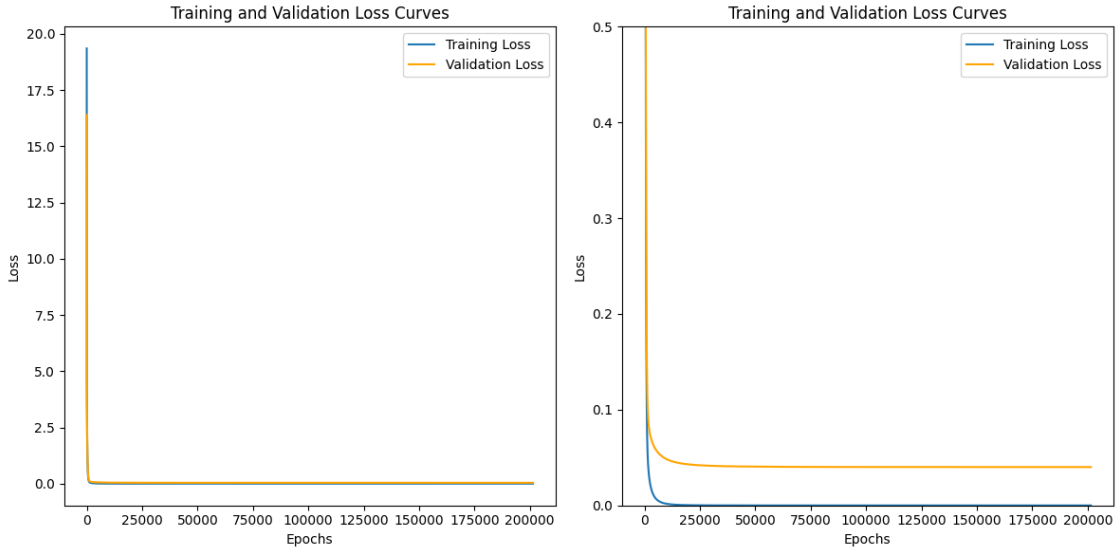
In this experiment the hidden size is 1000, norm of input values is 15. For training 100 points and validation 30 points are randomly generated. From the graph it can be observed that the difference between loss curves is around 0.05 - 0.1 until the end of training.

[4]:
```
model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=1000)
x_5_points = utils.generate_x_training_points(15,10,100)
x_3_points_val=utils.generate_x_training_points(15,10,30)
optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
initial_state_dict = copy.deepcopy(model.state_dict())
loss_function = nn.MSELoss()
epochs = 0
outputs = []
losses = []
```

```
training_losses = []
validation_losses = []
while True:
    epochs+=1
    reconstructed = model(x_5_points)
    loss = loss_function(reconstructed, x_5_points)
    optimizer.zero_grad()
    loss.backward()
    training_losses.append(loss.detach().numpy())
    optimizer.step()
    with torch.no_grad():
        model.eval()
        validation_reconstructed = model(x_3_points_val)
        validation_loss = loss_function(validation_reconstructed,␣
 ↪x_3_points_val)
        validation_losses.append(validation_loss.item())
    if loss<1e-7:

        trained_state_dict = model.state_dict()
        eigen_values_p5_1,diff_1 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[0])
        eigen_values_p5_2,diff_2 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[1])
        eigen_values_p5_3,diff_3 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
        eigen_values_p5_4,diff_4 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[3])
        eigen_values_p5_5,diff_5 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[4])
        eigen_values_p5    = torch.
 ↪cat((eigen_values_p5_1,eigen_values_p5_2,eigen_values_p5_3,eigen_values_p5_4,eigen_values_p
        eigen_values_normed_5p = torch.abs(eigen_values_p5)
        break
# Create subplots for side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# First subplot for training loss
epochs = list(range(1, len(training_losses) + 1))
ax1.plot(epochs, training_losses, label='Training Loss')
ax1.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')

ax1.set_title('Training and Validation Loss Curves')
ax1.legend()
```

```
# Second subplot for validation loss (same as the first subplot)
ax2.plot(epochs, training_losses, label='Training Loss')
ax2.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.set_ylim(0, 0.5)
ax2.set_title('Training and Validation Loss Curves')
ax2.legend()

plt.tight_layout()
plt.show()
```

```
/home/omar/Desktop/TUM/4Semester/Practical_course/utils.py:473: UserWarning: The
use of `x.T` on tensors of dimension other than 2 to reverse their shape is
deprecated and it will throw an error in a future release. Consider `x.mT` to
transpose batches of matrices or `x.permute(*torch.arange(x.ndim - 1, -1, -1))`
to reverse the dimensions of a tensor. (Triggered internally at
../aten/src/ATen/native/TensorShape.cpp:3571.)
  w_alpha = torch.matmul(temp,input_.T)
```



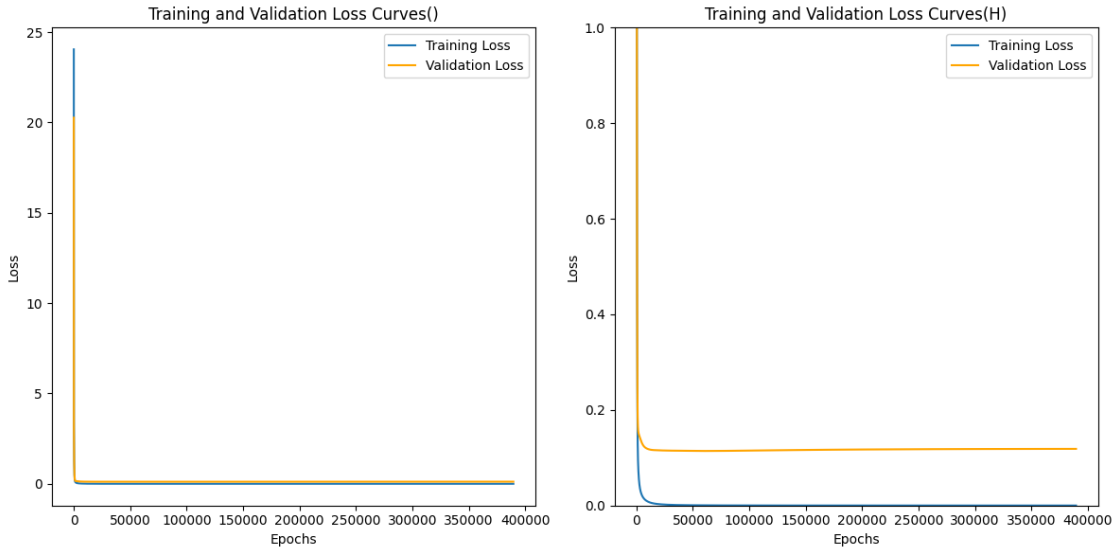
9.2 Hidden size 100 - input norm 15 - 100 training points

In this experiment the hidden size is chosen to be 100 instead of 1000 and the difference is now around 0.2.

[6]:
```
model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=100)
x_5_points = utils.generate_x_training_points(15,10,100)
x_3_points_val=utils.generate_x_training_points(15,10,30)
optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
```

```
initial_state_dict = copy.deepcopy(model.state_dict())
loss_function = nn.MSELoss()
epochs = 0
outputs = []
losses = []
training_losses = []
validation_losses = []
while True:
    epochs+=1
    reconstructed = model(x_5_points)
    loss = loss_function(reconstructed, x_5_points)
    optimizer.zero_grad()
    loss.backward()
    training_losses.append(loss.detach().numpy())
    optimizer.step()
    with torch.no_grad():
        model.eval()
        validation_reconstructed = model(x_3_points_val)
        validation_loss = loss_function(validation_reconstructed,␣
 ↪x_3_points_val)
        validation_losses.append(validation_loss.item())
    if loss<1e-6:
        trained_state_dict = model.state_dict()
        eigen_values_p5_1,diff_1 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[0])
        eigen_values_p5_2,diff_2 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[1])
        eigen_values_p5_3,diff_3 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
        eigen_values_p5_4,diff_4 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[3])
        eigen_values_p5_5,diff_5 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[4])
        eigen_values_p5    = torch.
 ↪cat((eigen_values_p5_1,eigen_values_p5_2,eigen_values_p5_3,eigen_values_p5_4,eigen_values_p
        eigen_values_normed_5p = torch.abs(eigen_values_p5)
        break
# Create subplots for side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# First subplot for training loss
epochs = list(range(1, len(training_losses) + 1))
ax1.plot(epochs, training_losses, label='Training Loss')
ax1.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')
```

```
ax1.set_title('Training and Validation Loss Curves()')
ax1.legend()

# Second subplot for validation loss (same as the first subplot)
ax2.plot(epochs, training_losses, label='Training Loss')
ax2.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.set_ylim(0, 1)
ax2.set_title('Training and Validation Loss Curves(H)')
ax2.legend()

plt.tight_layout()
plt.show()
```



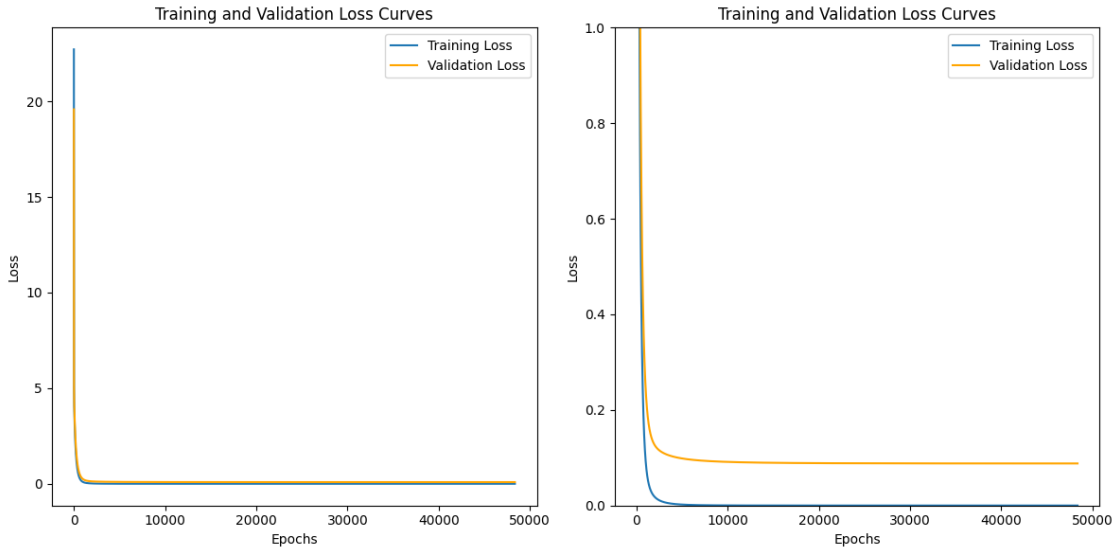
9.3 Hidden size 1000 - Norm 15 - 50 train points

In this experiment 50 training points is used instead of 100 and the results are similar as in the above experiment.

[7]:
```
model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=1000)
x_5_points = utils.generate_x_training_points(15,10,50)
x_3_points_val=utils.generate_x_training_points(15,10,20)
optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
initial_state_dict = copy.deepcopy(model.state_dict())
loss_function = nn.MSELoss()
epochs = 0
outputs = []
```

```
losses = []
training_losses = []
validation_losses = []
while True:
    epochs+=1
    reconstructed = model(x_5_points)
    loss = loss_function(reconstructed, x_5_points)
    optimizer.zero_grad()
    loss.backward()
    training_losses.append(loss.detach().numpy())
    optimizer.step()
    with torch.no_grad():
        model.eval()
        validation_reconstructed = model(x_3_points_val)
        validation_loss = loss_function(validation_reconstructed,␣
 ↪x_3_points_val)
        validation_losses.append(validation_loss.item())
    if loss<1e-7:

        trained_state_dict = model.state_dict()
        eigen_values_p5_1,diff_1 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[0])
        eigen_values_p5_2,diff_2 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[1])
        eigen_values_p5_3,diff_3 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
        eigen_values_p5_4,diff_4 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[3])
        eigen_values_p5_5,diff_5 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[4])
        eigen_values_p5    = torch.
 ↪cat((eigen_values_p5_1,eigen_values_p5_2,eigen_values_p5_3,eigen_values_p5_4,eigen_values_p
        eigen_values_normed_5p = torch.abs(eigen_values_p5)
        break
# Create subplots for side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# First subplot for training loss
epochs = list(range(1, len(training_losses) + 1))
ax1.plot(epochs, training_losses, label='Training Loss')
ax1.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')

ax1.set_title('Training and Validation Loss Curves')
ax1.legend()
```

```
# Second subplot for validation loss (same as the first subplot)
ax2.plot(epochs, training_losses, label='Training Loss')
ax2.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.set_ylim(0, 1)
ax2.set_title('Training and Validation Loss Curves')
ax2.legend()

plt.tight_layout()
plt.show()
```



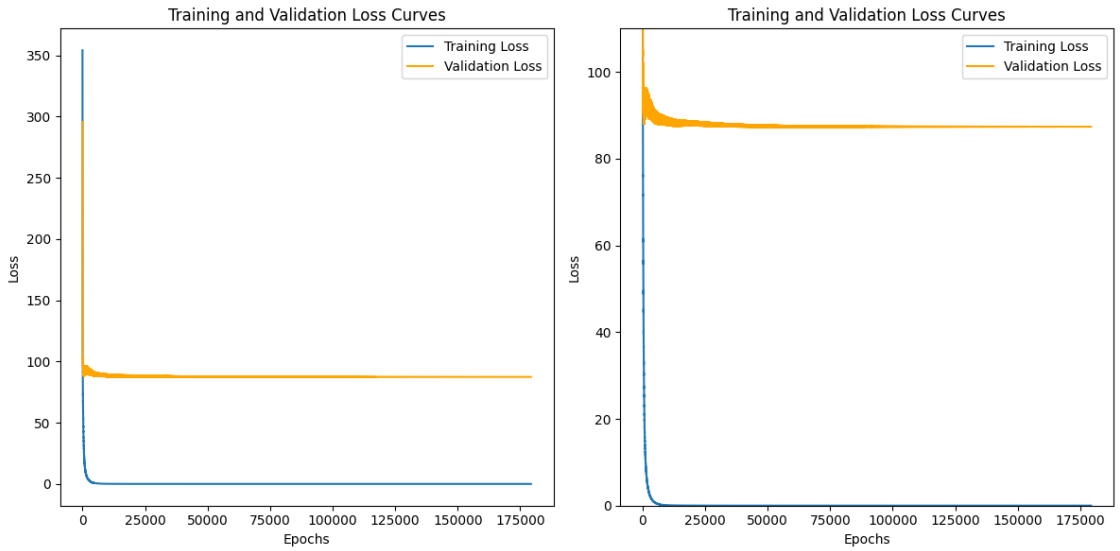
9.4 Mixed Norms

Here the input norms are randomly sampled between 2 to 100. The training loss reaches predefined threshold however the generalization gap is aroung 80-100. Based on this extension we can say that attractor networks can not generalize well.

[10]:
```
# Define parameters
num_samples = 50
dim = 10
radius_values = np.arange(2, 102, 2)

# Generate the training tensor
training_tensor = torch.cat([utils.sample_input_(radius, dim) for radius in␣
 ↪radius_values])
# Generate validation tensor
validation_radius_values = np.arange(2, 102, 4)
```

```
validation_tensor = torch.cat([utils.sample_input_(radius, dim) for radius in␣
 ↪validation_radius_values[:30]])

# Print the shape of the training tensor
model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=1000)
x_5_points =  torch.cat([utils.sample_input_(radius, dim) for radius in␣
 ↪radius_values])
x_3_points_val= torch.cat([utils.sample_input_(radius, dim) for radius in␣
 ↪validation_radius_values[:30]])
optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
initial_state_dict = copy.deepcopy(model.state_dict())
loss_function = nn.MSELoss()
epochs = 0
outputs = []
losses = []
training_losses = []
validation_losses = []
while True:
    epochs+=1
    reconstructed = model(x_5_points)
    loss = loss_function(reconstructed, x_5_points)
    optimizer.zero_grad()
    loss.backward()
    training_losses.append(loss.detach().numpy())
    optimizer.step()
    with torch.no_grad():
        model.eval()
        validation_reconstructed = model(x_3_points_val)
        validation_loss = loss_function(validation_reconstructed,␣
 ↪x_3_points_val)
        validation_losses.append(validation_loss.item())
    if loss<1e-7:

        trained_state_dict = model.state_dict()
        eigen_values_p5_1,diff_1 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[0])
        eigen_values_p5_2,diff_2 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[1])
        eigen_values_p5_3,diff_3 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
        eigen_values_p5_4,diff_4 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[3])
        eigen_values_p5_5,diff_5 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[4])
        eigen_values_p5    = torch.
 ↪cat((eigen_values_p5_1,eigen_values_p5_2,eigen_values_p5_3,eigen_values_p5_4,eigen_values_p
```

```
        eigen_values_normed_5p = torch.abs(eigen_values_p5)
        break
# Create subplots for side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# First subplot for training loss
epochs = list(range(1, len(training_losses) + 1))
ax1.plot(epochs, training_losses, label='Training Loss')
ax1.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')

ax1.set_title('Training and Validation Loss Curves')
ax1.legend()

# Second subplot for validation loss (same as the first subplot)
ax2.plot(epochs, training_losses, label='Training Loss')
ax2.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.set_ylim(0, 110)
ax2.set_title('Training and Validation Loss Curves')
ax2.legend()

plt.tight_layout()
plt.show()
```

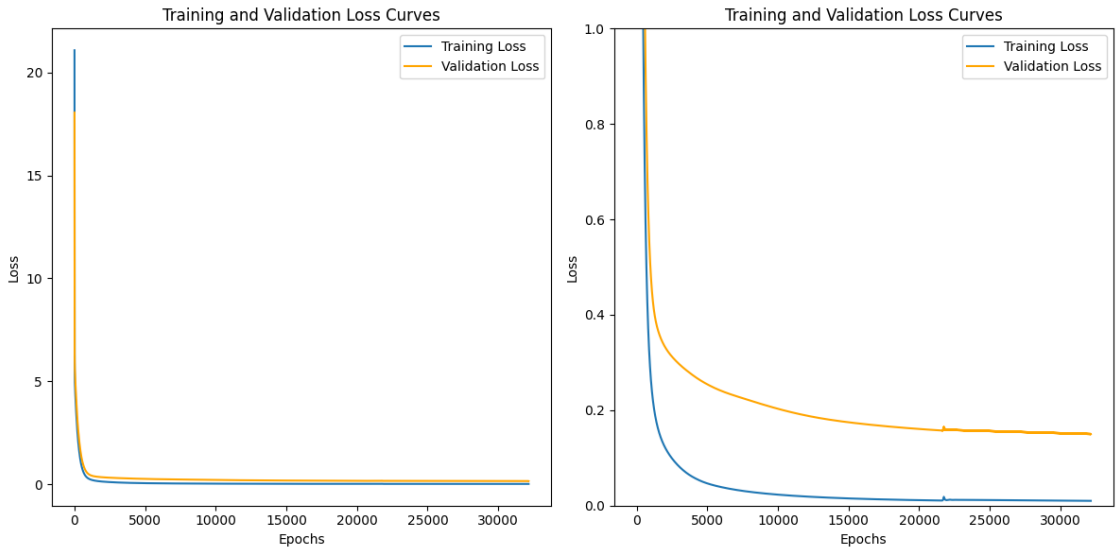


9.5 Non Attractor

For comparison a non-Attractor network is also trained. Here the training loss threshold is 10^-2 and that's why the curves in the graph are different to the previous ones.

[11]:
```
model = utils.Autoencoder_2_layers_(input_dim=10,hidden_dim=50)
x_5_points = utils.generate_x_training_points(15,10,100)
x_3_points_val=utils.generate_x_training_points(15,10,30)
optimizer = torch.optim.SGD(params=model.parameters(),lr=1)
initial_state_dict = copy.deepcopy(model.state_dict())
loss_function = nn.MSELoss()
epochs = 0
outputs = []
losses = []
training_losses = []
validation_losses = []
while True:
    epochs+=1
    reconstructed = model(x_5_points)
    loss = loss_function(reconstructed, x_5_points)
    optimizer.zero_grad()
    loss.backward()
    training_losses.append(loss.detach().numpy())
    optimizer.step()
    with torch.no_grad():
        model.eval()
        validation_reconstructed = model(x_3_points_val)
        validation_loss = loss_function(validation_reconstructed,␣
 ↪x_3_points_val)
        validation_losses.append(validation_loss.item())
    if loss<1e-2:

        trained_state_dict = model.state_dict()
        eigen_values_p5_1,diff_1 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[0])
        eigen_values_p5_2,diff_2 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[1])
        eigen_values_p5_3,diff_3 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[2])
        eigen_values_p5_4,diff_4 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[3])
        eigen_values_p5_5,diff_5 = utils.
 ↪calculate_jacobian_2_layers_(initial_state_dict,trained_state_dict,x_5_points[4])
        eigen_values_p5    = torch.
 ↪cat((eigen_values_p5_1,eigen_values_p5_2,eigen_values_p5_3,eigen_values_p5_4,eigen_values_p
        eigen_values_normed_5p = torch.abs(eigen_values_p5)
        break
```

```python
# Create subplots for side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# First subplot for training loss
epochs = list(range(1, len(training_losses) + 1))
ax1.plot(epochs, training_losses, label='Training Loss')
ax1.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')

ax1.set_title('Training and Validation Loss Curves')
ax1.legend()

# Second subplot for validation loss (same as the first subplot)
ax2.plot(epochs, training_losses, label='Training Loss')
ax2.plot(epochs, validation_losses, label='Validation Loss', color='orange')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.set_ylim(0, 1)
ax2.set_title('Training and Validation Loss Curves')
ax2.legend()

plt.tight_layout()
plt.show()
```



[ ]: